Software Fault Tolerance

EECE 513: Design of Fault-tolerant Digital Systems

(Based on ECE695B at Purdue Univ. by Prof. Saurabh Bagchi – used with permission)

Learning Objectives

- Define Software Fault-tolerance and enumerate its challenges
- Apply Process Pairs for Software Fault Tolerance
- List three diversity-based techniques and evaluate their respective pros and cons
- Use robust data structures for structural integrity

What is Software Fault Tolerance?

- Three alternative definitions

1. Management of faults originating from defects in design or implementation of software components
2. Management of hardware failures in software
3. Management of network failures

We will follow the classical definition (1) due to Avizienis in 1977

Motivation: Software Fault Tolerance

- Usual method of software reliability is fault avoidance using good software engineering methodologies
- Large and complex systems ⇒ fault avoidance not successful
  - Rule of thumb fault density in software is 10-50 per 1,000 lines of code for good software and 1-5 after intensive testing using automated tools
- Redundancy in software needed to detect, isolate, and recover from software failures
- Hardware fault tolerance easier to assess
- Software is difficult to prove correct

HARDWARE FAULTS	SOFTWARE FAULTS
1. Faults time-dependent	Faults time-invariant
2. Duplicate hardware detects	Duplicate software not effective
3. Random failure is main cause	Complexity is main cause

Consequences of Software Failure

- General Accounting Office reports $4.2 million lost annually due to software errors
- Launch failure of Mariner I (1962)
- Destruction of French satellite (1988)
- Problems with Space Shuttle and Apollo missions
- SS7 (signaling system) protocol implementation - untested patch (mistyped character) (1997)
- Therac 25 (overdose of medical radiation 1000's of rads in excess of prescribed dosage)
- Toyota Prius recall (2004) due to bug in embedded code

Difficulties

- Improvements in software development methodologies reduce the incidence of faults, yielding fault avoidance
- Need for test and verification
- Formal verification techniques, such as proof of correctness, can be applied to rather small programs
- Potential exists of faulty translation of user requirements
- Conventional testing is hit-or-miss. “Program testing can show the presence of bugs but never show their absence,” - Dijkstra, 1972.
- There is a lack of good fault models for software defects

Forms of Software Testing

- Exhaustive testing of reasonable sized applications is impossible
- Approach is to define equivalence classes of inputs so that only one test case from each class suffices
- Techniques proposed include
  - Path testing
  - Branch testing
  - Interface testing
  - Special values testing
  - Functional testing
  - Anomaly analysis
- Studies have shown path testing and interface testing while difficult to design afford good coverage for large number of applications

Software Fault Tolerance: Terms

- **ROBUSTNESS**: The extent to which software continues to operate despite introduction of invalid inputs.

  Example: 1. Check input data

  =>ask for new input

  =>use default value and raise flag

  2. Self checking software

- **FAULT CONTAINMENT**: Faults in one module should not affect other modules.

  Example: Reasonable checks

  Watchdog timers

  Overflow/divide-by-zero detection

  Assertion checking

- **FAULT TOLERANCE**: Provides uninterrupted operation in presence of program fault through multiple implementations of a given function

Features of software faults

- Mature software exhibits nearly constant failure rate
  - Bathtub curve for modeling entire lifetime from release to retirement
- Number of failures is correlated with
  - Execution time
  - Code density
  - Software timing,
  - Synchronization points

Temporal Redundancy

- Reexecution of a program when error is encountered
- Error may be faulty data, faulty execution or incorrect output
- Reexecution will clear errors arising from temporary circumstances
- Examples: Noisy communication channel, Full buffers, Power supply transients, Resource exhaustion in multiprocess environment
- Provides fault containment
- Possible to apply to applications with loose time constraints

Multi-Version Software Fault Tolerance

- Use of multiple versions (or “variants”) of a piece of software
- Different versions may execute in parallel or in sequence
- Rationale is that multiple versions will fail differently, i.e., for different inputs
- Versions are developed from common specifications
- Three main approaches
  - Recovery Blocks
  - N-version Programming
  - N Self-Checking Programming

Learning Objectives

- Define Software Fault-tolerance and enumerate its challenges
- Apply Process Pairs for Software Fault Tolerance
- List three diversity-based techniques and evaluate their respective pros and cons
- Use robust data structures for structural integrity

Process Pairs

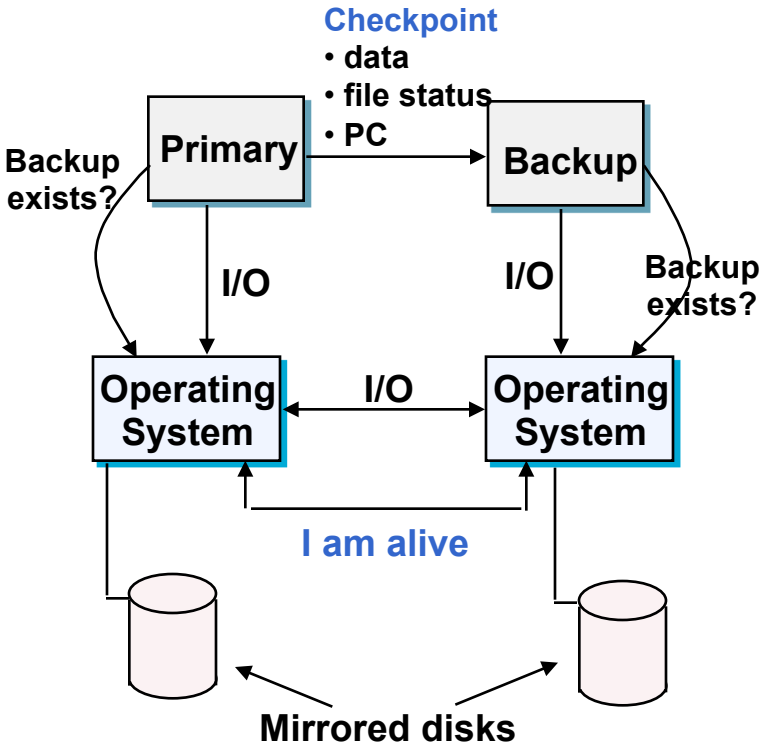
- Used in HP Himalaya servers as part of their NonStop Advanced Architecture
- Bragging rights of the architecture
  - Run the majority of credit and debit card systems in N.America
  - More than US$3 billion of electronic funds transfers daily
  - Run many of the E911 systems in North America
- Primary and backup processes on two different processors
- Primary process executes actively
  - Backup process is kept current by periodically sending state of primary process
- Processors execute fail-stop failure
  - When processor failure detected, backup takes over

Process Pairs

- **Applicability**
  - Permanent and transient hardware and software failures
  - Loosely coupled redundant architectures
  - Message passing process communication
  - Well suited for maintaining data integrity in a transactional type of system
  - Can be used to replicate a critical system function or user application

- **Assumptions**
  - Hardware and software modules design to *fail-fast*, i.e., to rapidly detect errors and subsequently terminate processing
  - Errors can be corrected by re-executing the same software copy in changed environment

Process Pairs Mechanism in Tandem Guardian OS



1. The application executes as *Primary*
2. *Primary* starts a *Backup* on another processor
3. Duplicated file images are also created
4. *Primary* periodically sends checkpoint information to *Backup*
5. *Backup* reads checkpoint messages and updates its data, file status, and program counter
   - the checkpoint information is inserted in the corresponding memory locations of the *Backup*
7. *Backup* loads and executes if the system reports that *Primary* processor is down
   - the error detection is done by *Primary* OS or
   - *Primary* fails to respond to *"I am alive"* message
8. All file activities by *Primary* are performed on both the primary and backup file copies
9. *Primary* periodically asks the OS if a *Backup* exists
   - if there is no *Backup*, the *Primary* can request the creation of a copy of both the process and file structure

Evaluation of Process-Pairs

- Done for Tandem's Guardian OS Studied Tandem Product Report (TPR) which are used to report product failures
- Problem classified as software fault only after analysts have pinpointed the cause
- Classes of software faults (not exhaustive)
  - Incorrect computation (3%)
  - Data fault (15%)
  - Missing operation (20%)
  - Side effect of code update (4%)
  - Unexpected situation (29%)
  - Microcode defect (4%)

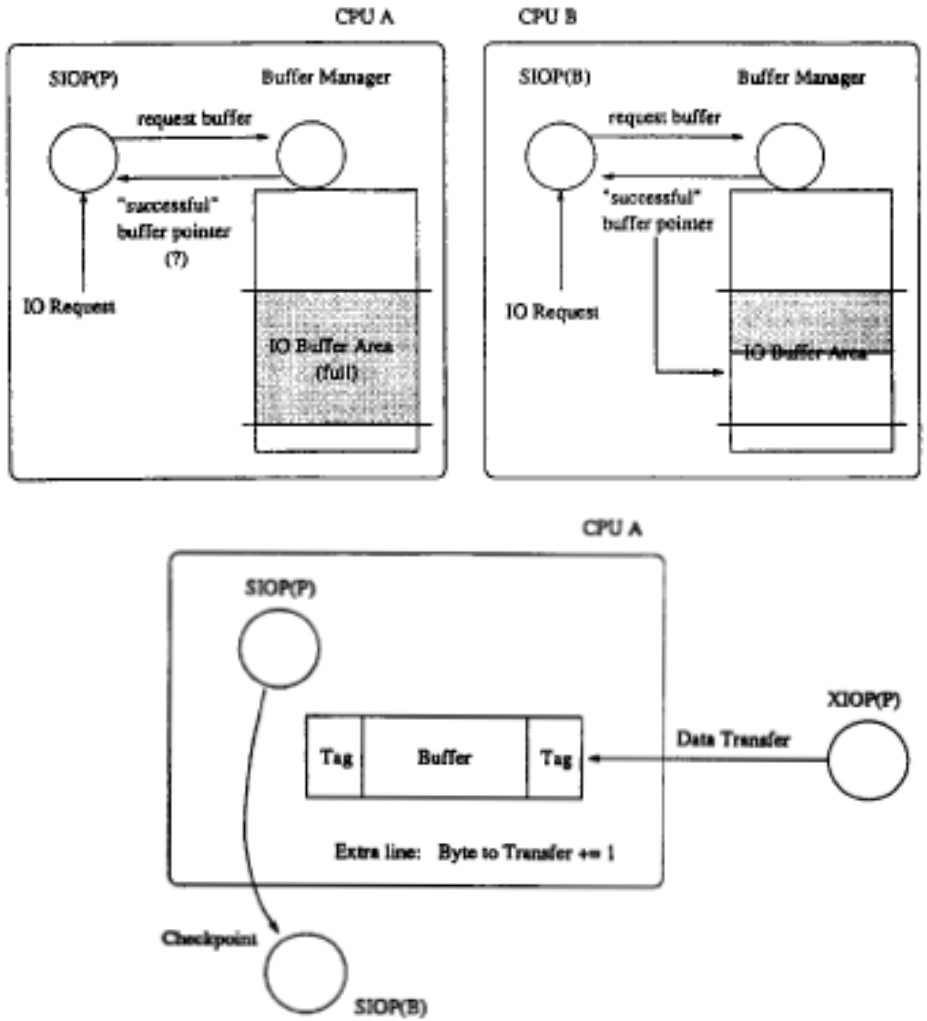
Results from Evaluation

- Out of total software failures, 138 out of 169 (82%) caused single processor halt (recoverable). This is a measure of the software fault tolerance of the system.
- Reasons for multiple processor fault
  - Same fault as in the primary: 17/28 (60%)
  - Second fault during job recovery: 4/28 (14.3%)
  - Second halt is not related to process pairs: 4/28 (14.3%)

Results from Evaluation

- Reasons for uncorrelated software fault
  - Backup reexecutes same task, but same fault not exercised: 29%.
    - Different memory state
    - Race or timing related problem
  - Example:
    - Privileged process on primary requests a buffer
    - Because of high user activity on primary, buffer exhaustion
    - Bug in buffer management routine and returns “success”
    - Primary privileged process uses uninitialized buffer pointer and causes processor halt
    - Backup process served the request after takeover
    - But buffer was available on the backup processor

Figure for Cases of Software Fault Tolerance



Results from Evaluation

- Reasons for uncorrelated software fault
  - Backup does not reexecute failed request on takeover: 20%.
    - Processor monitoring task
    - Interactive task
  - Effect of error latency: 5%
    - Task that caused the error finished before detection
    - Example: I/O process for copying buffer from source to destination.
    - Copied an additional byte overwriting buffer tag.
    - No problem in data transfer.
    - The successful data transfer was checkpointed but not the corrupted buffer tag
    - Problem surfaces later when buffer manager verifies buffer.
    - No problem when reexecuting on backup.

Results from Evaluation

- Process pairs with checkpointing and restart recovers from 75% of reported software faults that result in processor failures
- The complexity of process pairs introduces some faults
  - 16% of single processor halts were failures of backup processes
- Counter-intuitive result since same software run on both processors
- Loose coupling between processors, long error latency, operation using checkpoints and not lock-step
- Are process triples better than process pairs?

Process Pairs
Advantages & Disadvantages

Advantages

- Extremely successful in Tandem OLTP applications
- Tolerates hardware, operating system, and application failures
- High coverage (> 90%) of hardware and software faults
- The backup does not significantly reduce the performance

Disadvantages

- Necessity of error detection checks and signaling techniques to make a process fail-fast
- Process pairs are difficult to construct for non-transaction-based applications

Learning Objectives

- Define Software Fault-tolerance and enumerate its challenges
- Apply Process Pairs for Software Fault Tolerance
- List three diversity-based techniques and evaluate their respective pros and cons
- Use robust data structures for structural integrity

Diversity

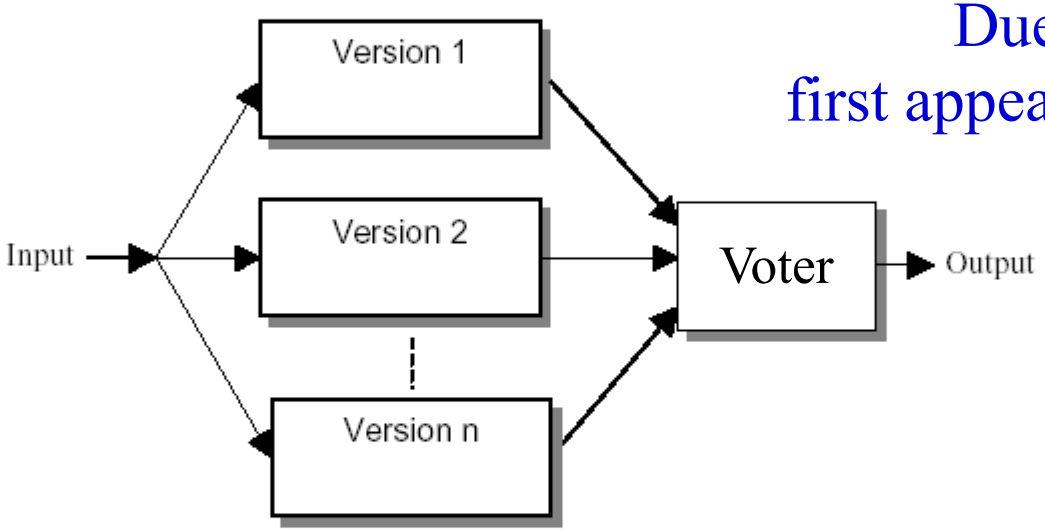
- Diversity as a technique for fault-tolerance goes back to the British Astronmer, Lord Maskelyne [Anh-2009]
  - Used two computers (human) to calculate lunar tables, when moon is at peak and its lowest point and compare the values
- Charles Babbage used Diversity in analytical engine

"When the formula to be computed is very complicated, it may be algebraically arranged for computation in two or more totally distinct ways, and two or more sets of cards may be made. If the same constants are now employed with each set, and if under these circumstances the results agree, we may be quite secure of the accuracy of them all."

Multi-Version Software Fault Tolerance

- Use of multiple versions (or "variants") of a piece of software
- Different versions may execute in parallel or in sequence
- Rationale is that multiple versions will fail differently, i.e., for different inputs
- Versions are developed from common specifications
- Three main approaches
  - N-version Programming
  - Recovery Blocks
  - N Self-Checking Programming

N-Version Programming



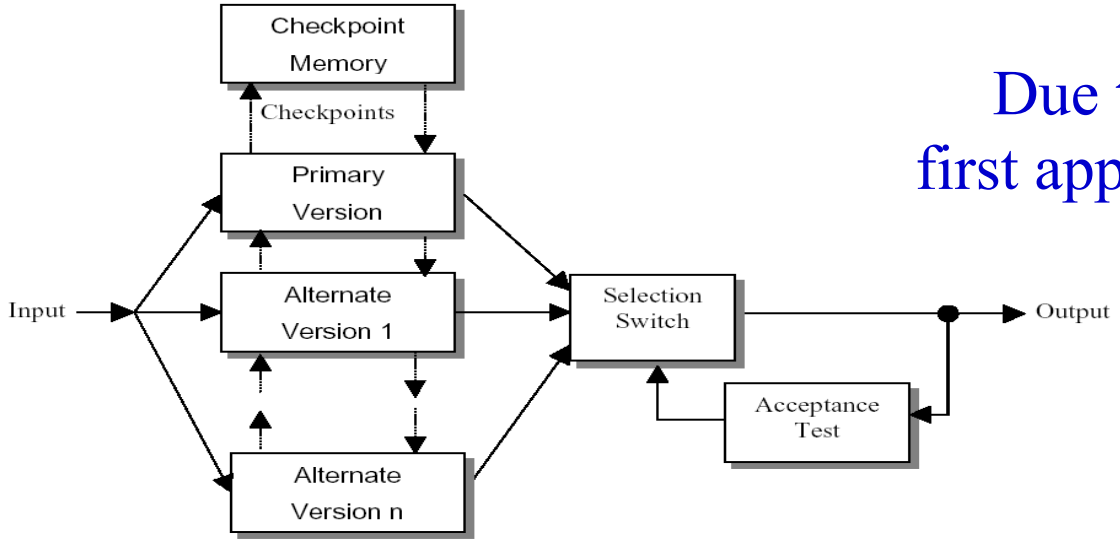
Due to Al Avizienis,
first appeared in CompSAC 1977

- All versions designed to satisfy same basic requirement
- Decision of output comparison based on voting
- Different teams build different versions to avoid correlated failures

Pros and Cons of NVP

- NVP relies on independence among the versions
  - But not always true in practice [Knight and Leveson'83]
- Why does this happen ?
  - People make same mistakes, e.g., incorrect treatment of boundary conditions
  - Some parts of a problem are more difficult than others - similarity in programmer's view of "difficult" regions
  - Specifications may themselves be incorrect/incomplete
- Note: This does not mean NVP is useless. Rather, it does not always mean that NVP will detect S/W faults. Its reliability is upper-bounded by independence.

Recovery Blocks



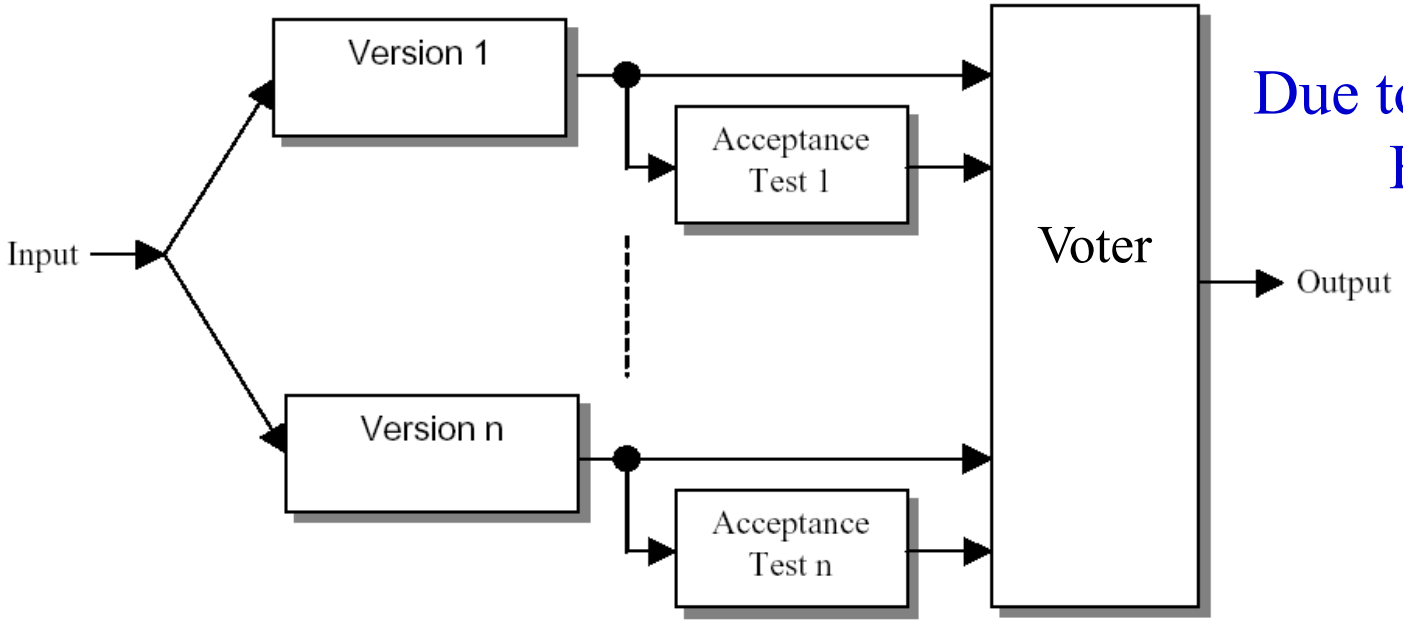
Due to Brian Randell, first appeared in ToSE 1975

- Checkpoint and restart approach
  - Try a version, if error detected through acceptance test, try a different version
  - Ordering of the different versions according to reliability
- Checkpoints needed to provide valid operational state for subsequent versions (hence, discard all updates made by a version)
- Acceptance test needs to be faster and simpler than actual code

Pros and Cons of RB

- Advantages
  - No performance or area overheads in the fault-free case, except the state saving overhead.
  - Allows gradual evolution of software components. Old versions can be replaced with new ones, and used as secondary.
  - Nice hierarchical design (structured approach)
- Disadvantages
  - Reliability depends on the coverage of the acceptance test. Acceptance test should be independent of the main version, but faster (e.g., range checks)
  - State saving mechanisms need to be employed.
  - Requires transaction-like semantics. Cannot always undo side-effects.

N Self-Checking Programming



Due to J. C. Laprie, FTCS 87

- Multiple software versions with structural variations of RB and NVP
- Use of separate acceptance tests for each version

Pros and Cons of NSCP

- Advantages
  - Combines advantages of NVP and RBs
  - Ensure that some errors are caught before the voting stage
  - Provides error containment
  - Almost no disruption in service due to faults
- Cons
  - Incurs more overhead than NVP and ‘N’ times the overhead of RB
  - Does not protect against errors in specifications
  - Extra effort to derive acceptance tests and write the N-versions

Similarity to H/W Fault-tolerance

- RB is equivalent to the stand-by sparing (of passive dynamic redundancy)
- NVP is equivalent to N-modular redundancy (static redundancy)
- NSCP is equivalent to active dynamic redundancy. A self-checking component results either from:
  - Association of an acceptance test to a version
  - Association of two variants with a comparison algorithm

Reliability Analysis of Multi-Version Approaches

Three postulates of software development [Sha-2000, IEEE Software]

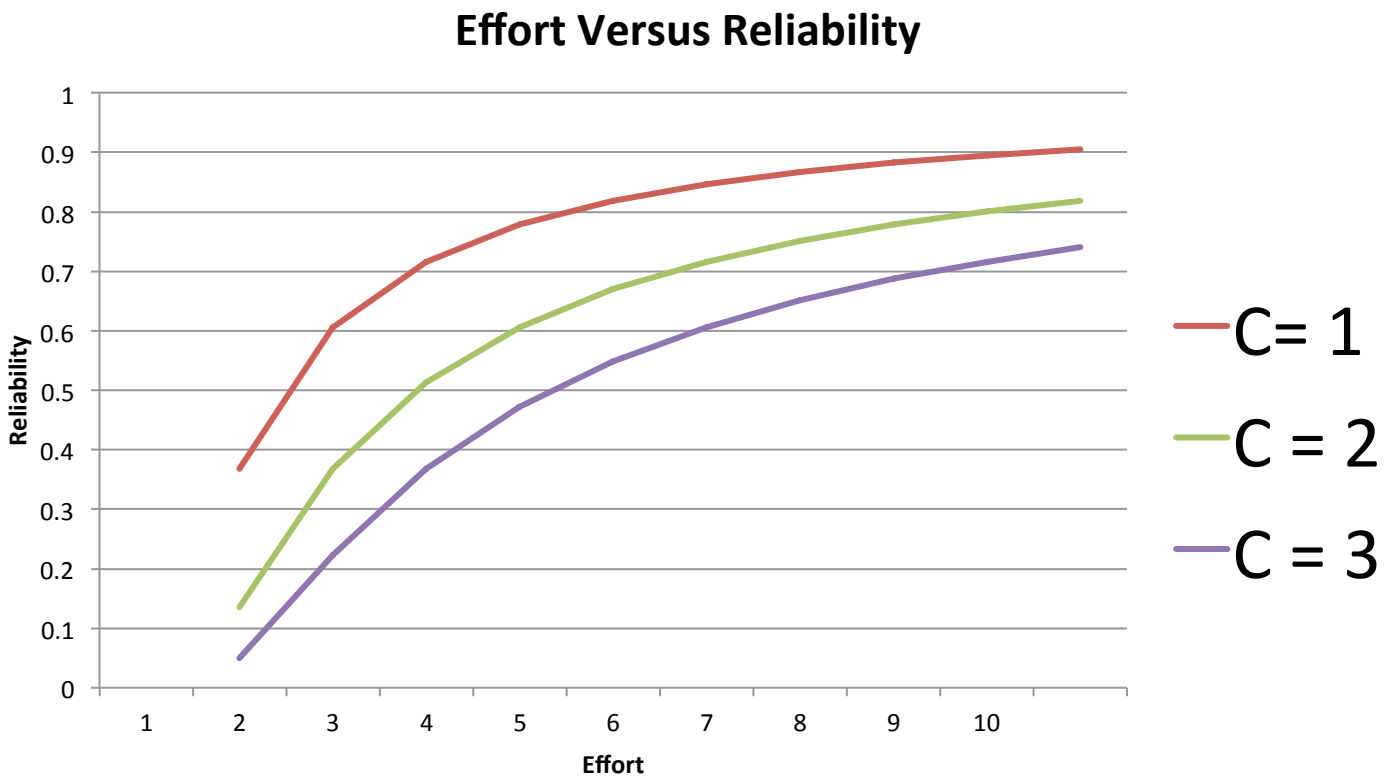
P1: Complexity Breeds Bugs: Everything else being equal, the more complex the software project is, the harder it is to make it reliable.

P2: All Bugs are Not Equal: You fix a bunch of obvious bugs quickly, but finding and fixing the last few bugs is much harder, if you can ever hunt them down.

P3: All Budgets are Finite: There is only a finite amount of effort (budget) that we can spend on any project. That is, if we go for n version diversity, we must divide the available effort n-way.

- We attempt to analyze the reliability of the three systems using methods from combinatorial modeling. We assume the following:
  - $R(t) = e^{-\lambda t}$
  - Failure rate $\lambda \propto 1/\text{Effort (E)}$
  - Failure rate $\lambda \propto$ Complexity (C)
  - Let $\lambda = kC / E$

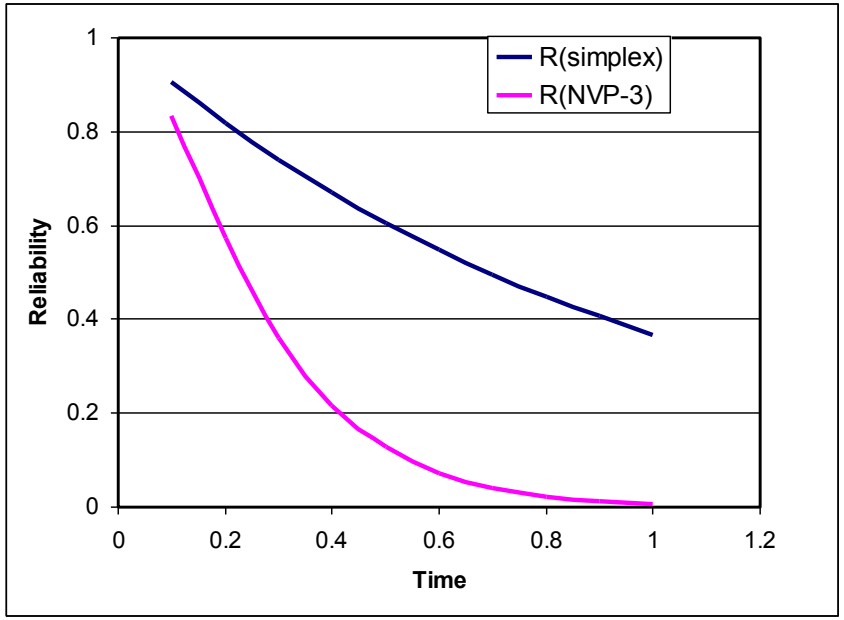
Reliability Vs Effort Vs Complexity



$$R_{simplex} = \exp(-kCt/E)$$

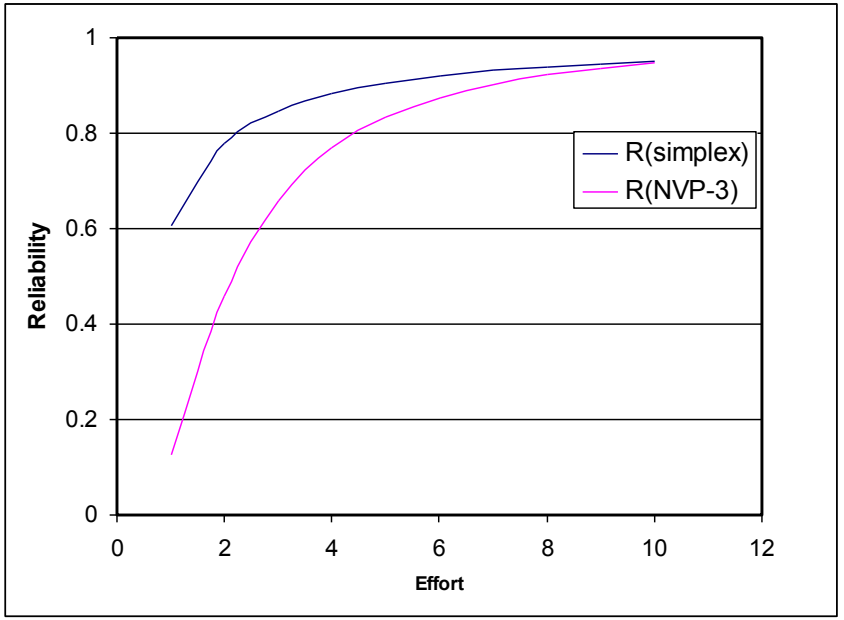
Reliability of NVP vs. single version

For Effort = 1



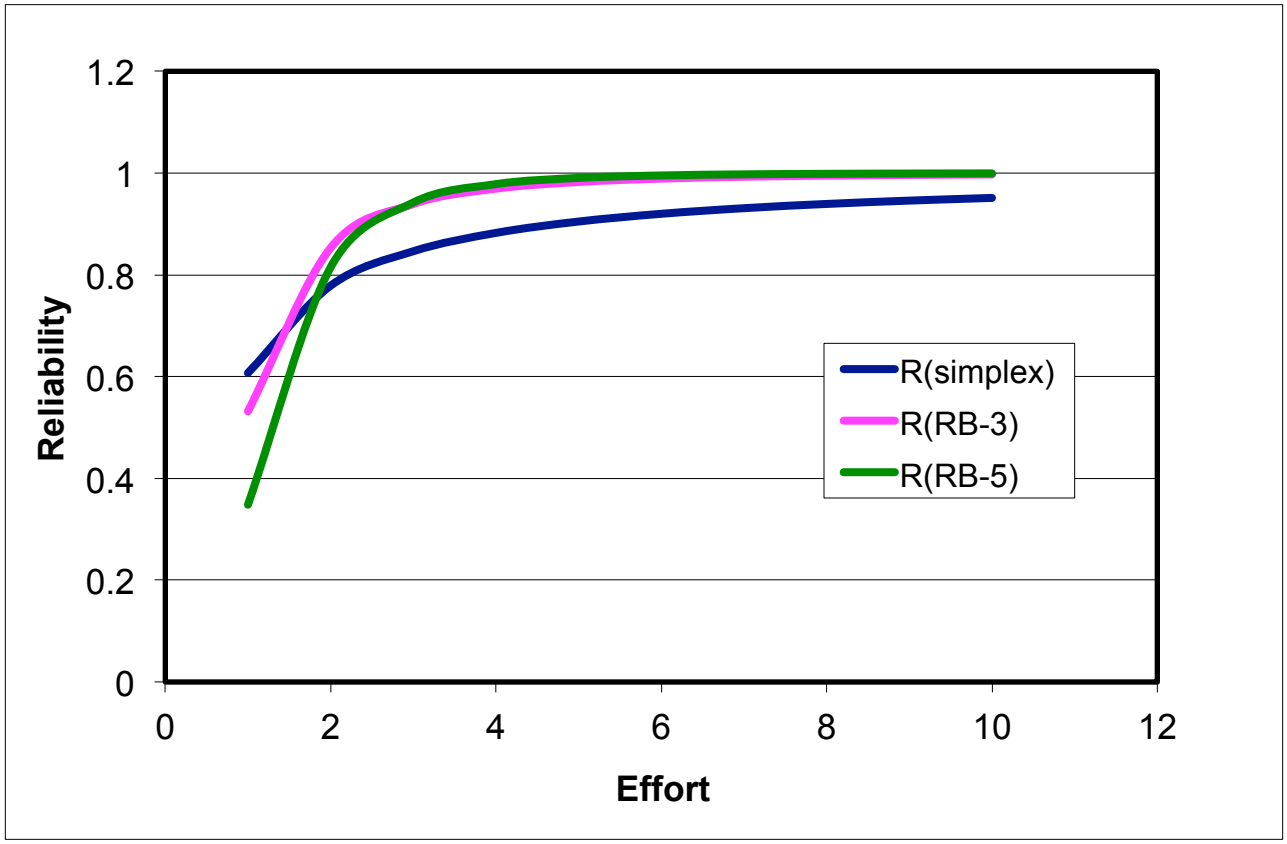
$$R_{simplex} = \exp(-kCt/E)$$

At time = 1



$$R_{NVP} = 3\exp(-6kCt/E) - 2\exp(-9kCt/E)$$

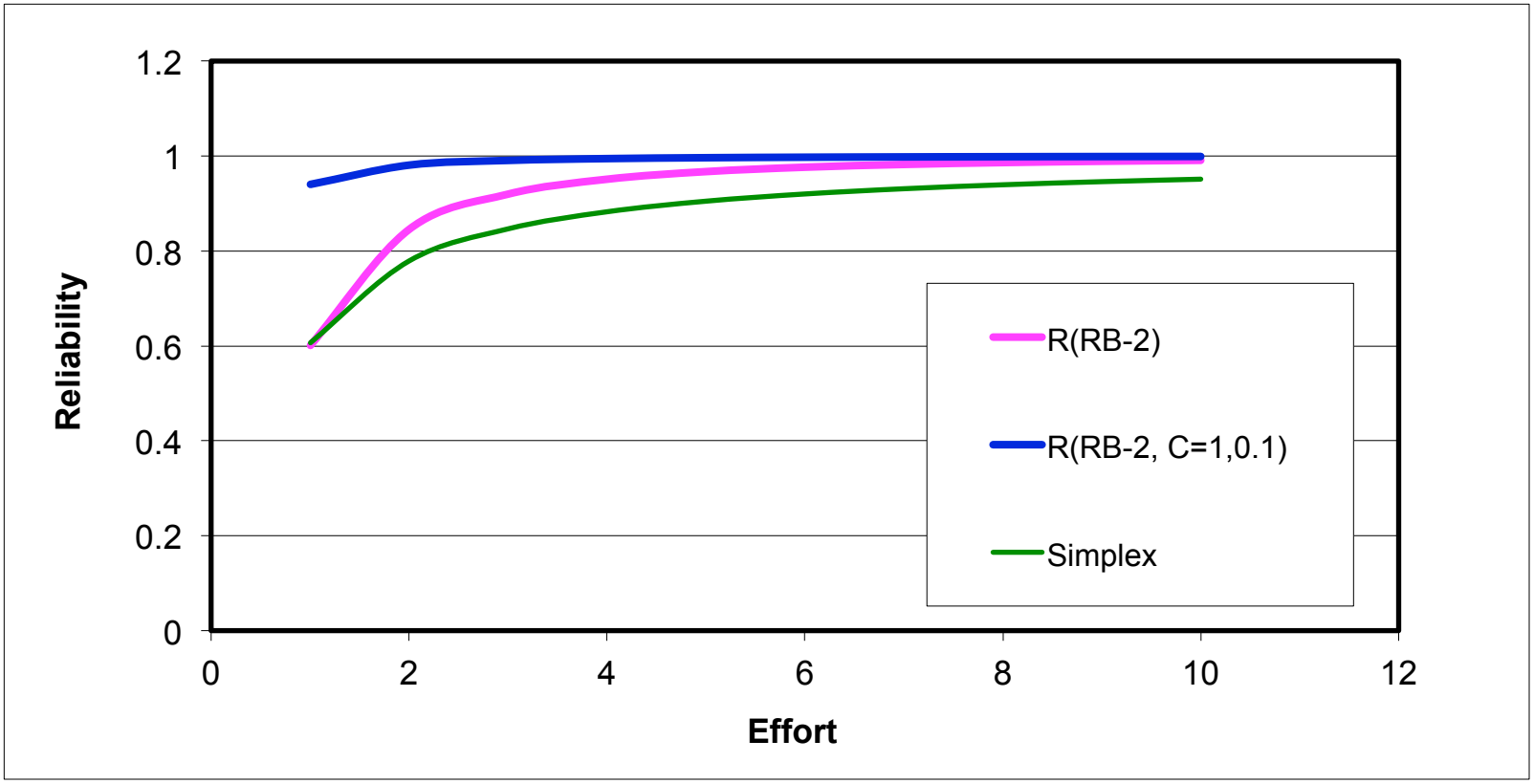
Reliability of RB vs. Simplex



$R_{simplex} = \exp(-kCt/E)$

$R_{RB} = 1-(1-\exp(-3kCt/E))^3$

Effort Vs. Complexity



Reducing complexity of alternatives drastically improves the availability. So using simpler alternates is good !

Diversity: Summary

- Simplicity often yields higher benefits than diversity and its associated complexity
- Given a choice between increasing effort for a single component and building > 2 diverse components with less effort, prefer the former
- Recovery blocks with simpler alternates do offer benefits over Simplex architecture

Learning Objectives

- Define Software Fault-tolerance and enumerate its challenges
- Apply Process Pairs for Software Fault Tolerance
- List three diversity-based techniques and evaluate their respective pros and cons
- Use robust data structures for structural integrity

Robust Data Structures: Goals

- The goal is to find storage structures that are robust in the face of errors and failures
- What do we want to preserve?

Semantic integrity - the data is not corrupted

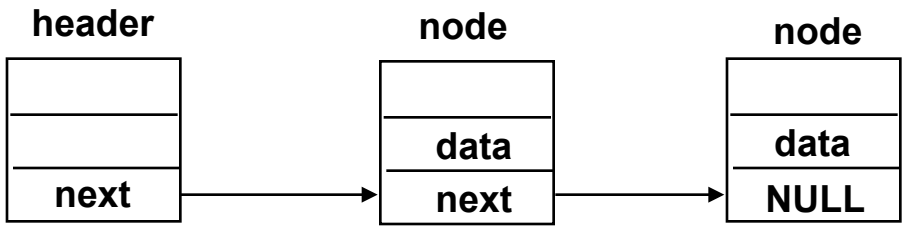
Structural integrity - the correct data representation is preserved

Robust Data Structure: Definition

- A robust data structure contains *redundant data* which allow *erroneous changes* to be detected, and corrected
  - a change is defined as an elementary (e.g., as single word) modification to the encoded form (e.g., data structure representation in memory) of a data structure instance
  - structural redundancy
    - a stored count of the numbers of nodes in a structure instance
    - identifier fields
    - additional pointers

Example: Linked Lists

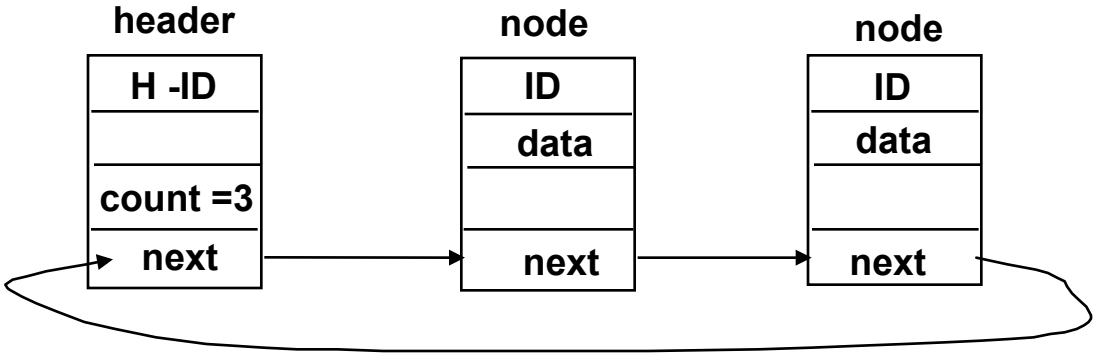
- Non-robust data structure: No redundant information to detect/recover from pointer errors



0-detectable and 0-correctable
changing one pointer to NULL can reduce any list to empty list

Example: Robust List

- Additions for improving robustness
  - an identifier field to each node
  - replace the NULL pointer in the last node by pointer to the header of the list
  - stores a count of the number of nodes



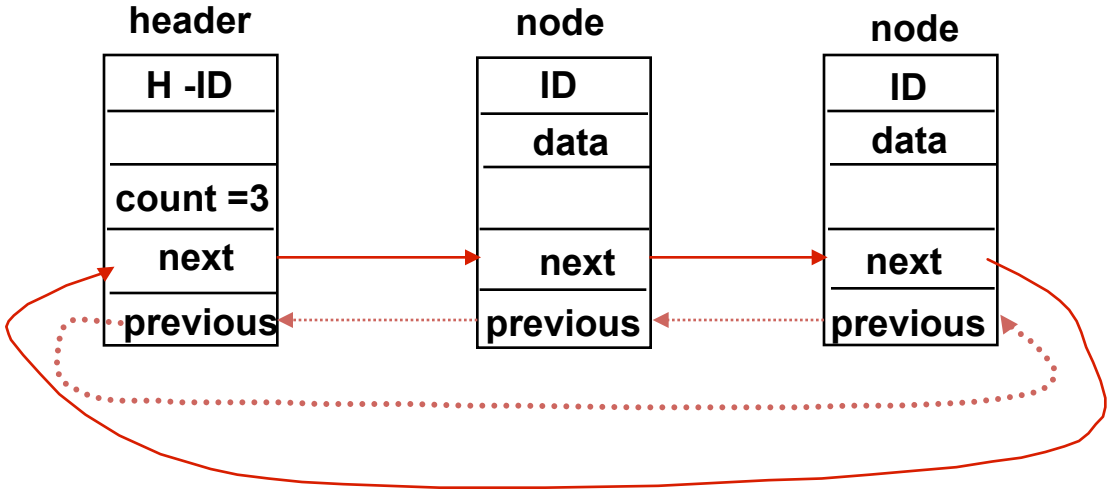
1-detectable and 0-correctable

•change to the count can be detected by comparing it against the number of nodes found by following pointers

•change to the pointer may be detected by a mismatch in count number or the new pointer points to a foreign node (which cannot have a valid identifier)

Example: Robust Double Linked List

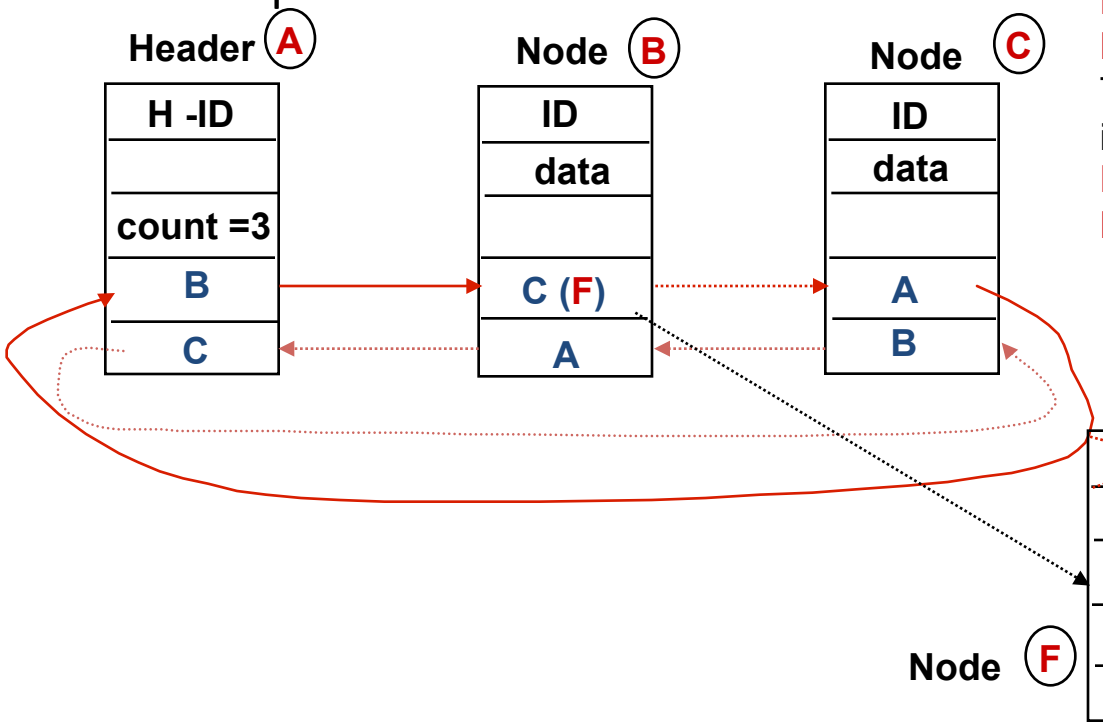
- Additions for improving robustness: Make it a double linked list



2-detectable and 1-correctable
the data structure has two independent, disjoint sets of pointers, each of which may be used to reconstruct the entire list

Error Correcting in Double-Linked List

- *Scan the list in the forward direction* until an identifier field error or forward/backward pointer mismatch is detected
- When this happens *scan the list in the reverse direction* until a similar error is detected
- Repair the data structure



The forward scan detects a mismatch in Node B and sets
$Local\_Ptr_B$ = B (local node's pointer)
$Next\_Ptr_B$ = F (pointer to the next node)
The reverse scan detects a mismatch in Node C and sets
$Local\_Ptr_C$ = C (local node's pointer)
$Back\_Ptr_C$ = B (pointer to the previous node)

Correction
($Local\_Ptr_B == Back\_Ptr_C$) ⇒
$Next\_Ptr_B := Local\_Ptr_C$
i.e., ($Next\_Ptr_B$ = C)

Robust Data Structures: Summary

- Advantages
  - Incurs much lower overheads than full duplication
  - Can detect both S/W and H/W errors that corrupt DS
  - Independent of programming language/compiler
- Limitations
  - Not transparent to the application
  - Best in tolerating errors which corrupt the structure of the data (not the semantics)
  - Increased complexity of the update routines may make them error prone – error propagation

Learning Objectives

- Define Software Fault-tolerance and enumerate its challenges
- Apply Process Pairs for Software Fault Tolerance
- List three diversity-based techniques and evaluate their respective pros and cons
- Use robust data structures for structural integrity